Signals-based Testing of Microsoft Office with an AI-Driven Robot Army

Wayne Roseberry

Wayne_Roseberry@hotmail.com

Abstract

From 2017 to 2023, I led an effort to use machine learning and AI driven self-learning autonomous agents to test Microsoft Office. The goal was to get as much product coverage as we could under complex usage conditions and use the product signal data, telemetry emitted by the Office applications, to find errors and bugs not exposed by our other automation solutions or testing efforts. By allowing the AI agents to explore the application using UI controls only, we hoped to uncover conditions and sequences more closely approximating what users might experience.

This experience was productive, rewarding, fascinating and difficult. We did indeed find problems real users were hitting that other testing efforts did not discover. We demonstrated that in the race to data, we could discover problems before the first users were reporting them. We showed that self-learning automatons could cover large functional surface area in complex ways under long running sessions. We also learned that doing all of that took a lot of effort in guiding the agents, addressing testability issues in the product, building an approach and method for recognizing problems from the data stream, writing tools to help understand the results, and finally driving the social campaign to help product teams understand how the data was helpful.

This paper shares the story of that experience, starting with some background, briefly describing how the system worked, and then focusing primarily on the challenges that came from working with such a solution. The challenges are especially important to understand, because they are present with all classes of AI and ML driven testing.

Biography

Wayne Roseberry is an SDET Manager at Ford Motor Corporation.

Wayne worked at Microsoft Corporation from June of 1990 until March of 2023. He started with a four-year tenure in product support, taking customer phone calls. His testing career launched in April 1994 on the MS Research project that became MSN 1.0. He worked on several versions of MSN, Microsoft Commercial Internet Services, Site Server, and then as one of the founding members of the SharePoint Portal team. SharePoint moved into the Office team, where Wayne eventually joined the Office Engineering team which builds the tools and services that the Office team uses to build, deliver and test Microsoft Office. Most of Wayne's time in the 11+ years on the Office Engineering team were spent working on the automation system focusing on issues like delivery efficiency, automation suite reliability, integrating product telemetry signal with test automation processes, supporting long run and complex automation scenarios, and finally using ML to drive product automation.

Outside of work, Wayne likes spending time with his wife and three kids. He likes to paint, draw, write and play music. He blogs frequently about testing topics.

Copyright Wayne Roseberry, 2024

1 Introduction

I started this journey wondering about ways to exploit automation to cover types of testing that were difficult to do. One of the largest challenges in testing is covering complex sequences of operations in the UI space. On a large application there is more functional surface than a team of engineers can hope to cover themselves. Writing automation to cover it is difficult, and time consuming, and as the complexity of the scenarios grows so too does the difficulty of maintaining control and understanding of the application state.

I had already done a great deal of work integrating product telemetry signal into the test engineering process and was curious how that information could guide test automation efforts. This led me to an interest in machine learning and AI driven solutions.

I came up with an approach that I soon realized others had discovered. Collaborating with a team at Microsoft who built tools around the same problem, I unified my signals-based testing approach with a self-learning AI testing tool. I realized very quickly I was stepping into an entirely new way of testing an application. This new approach came with exciting possibilities, but a lot of very big challenges. We may imagine that a self-learning robot army solves all our problems, but the truth is the same as any other new tool. The added power comes with new difficulties, and in some ways is more difficult than what we were doing before.

2 Background

2.1 Signals-based testing

Signals-based testing is *when we exploit engineering and operations data emitted by the product in order to find bugs by applying some workload against the product.*

Engineering and operations data are those pieces of information created by the product which are meant for product team use rather than customer use. This might include product log files, used primarily for diagnostic purposes. In this paper, I am referring primarily to product telemetry, data sent by the product across the internet to the product team, which in this case is the Microsoft Office team.

The product team uses this data for multiple reasons. The primary business purpose is to analyze customer usage patterns to find ways to improve the product, produce new features, understand which features are used in which ways. Another purpose used extensively by the product team is to discover problems by looking for signs of error and failure in the telemetry stream. Bug fixed are selected using this data, and fixed bugs are tracked via this data after release to see if fixes were effective.

All this signal data was coming from end users, after the product had been released. It was my intent to use this same signal stream pre-release while testing the product. Around 2016, I made changes to the Microsoft Office client and our engineering systems to allow us to integrate the testing systems with the product telemetry signal in a way that allowed us to isolate test traffic from end users, as well as filter and identify on a per test basis which test was running at the time the data was collected. We began collecting telemetry data from our automation runs and applying similar analysis on that data as we were the customer generated signals.

It was after that when I decided to try using the telemetry signal to train AI driven test automation.

2.2 Using signals to train ML

Figure 1 PNSQC 2017 where I introduce the idea of using telemetry signal to train an ML agent.

During a presentation I gave at PNSQC 2017, I talked about using a telemetry signal to evaluate and study test automation. It was during that talk where I referenced examples of research on using reinforcement learning to train an ML agent to play Mario. I suggested during this presentation that it was feasible we could do the same thing with applications under test. Ironically, it was during this same PNCQC event that Jason Arbon was demonstrating a similar idea, Test.ai, which used supervised learning to help ML-guided bots build graphs of application functionality that could be played back later.

The distinguishing characteristic of my idea was that the telemetry signal, the same thing the product team used to track customer usage patterns, could be used as a reward mechanism for the training system. The agent could learn its way around the application, and we could tell it when it had hit some feature or behavior, we wanted it to remember. I already had experiments underway building such a system, but there was another team at Microsoft building the same thing. They had consulted with me, I told them about the idea to use telemetry signals to train the bots, and eventually they had a system working well enough I dropped my project and adopted theirs.

3 How the Robot Army Works

The team building the tool I used referred to it as a “robot army.” This is because it utilized many machines for a run, thousands. The system learns through random behavior, which is why so many machines are used. There are three steps: 1) exploration, 2) reward collection and training, 3) guided navigation based on the trained model.

An important point to understand is that the training does not ever attempt to assess whether product behavior is correct, expected, or even incorrect. The only goal of the tool is to teach an agent how to navigate through an application to reach specific states. The purpose for this goal is to keep the product under test “moving,” and to do so in ways that are both complex and targeting features we desire.

3.1 Exploration to build the state-action-state graph

A scheduled job launches multiple (thousands) of autonomous agents running on different machines. The agents can be configured to target specific applications or processed, or to avoid certain applications. This configuration allows the training to focus on a specific application (e.g., Word, Notepad, Calculator, Excel, etc.) instead of randomly navigating everything on the OS.

The agent seeks the target application via various mechanisms (applications menu, search bar, etc.) and launches it. Once the application has completed loading, the agent takes a snapshot of the UI state using the `System.Windows.Automation` namespace (the same one used to build accessibility tools). At that point it randomly selects an action based on the available and enabled controls and executes whatever that was (e.g. "Click Open"). The UI changes based on the action, the agent captures the changes and a mapping of the prior UI state, action taken, and resulting UI state to a service in the cloud. The agent repeats the prior sense, think, act pattern sending each UI state->action->UI state set to the service.

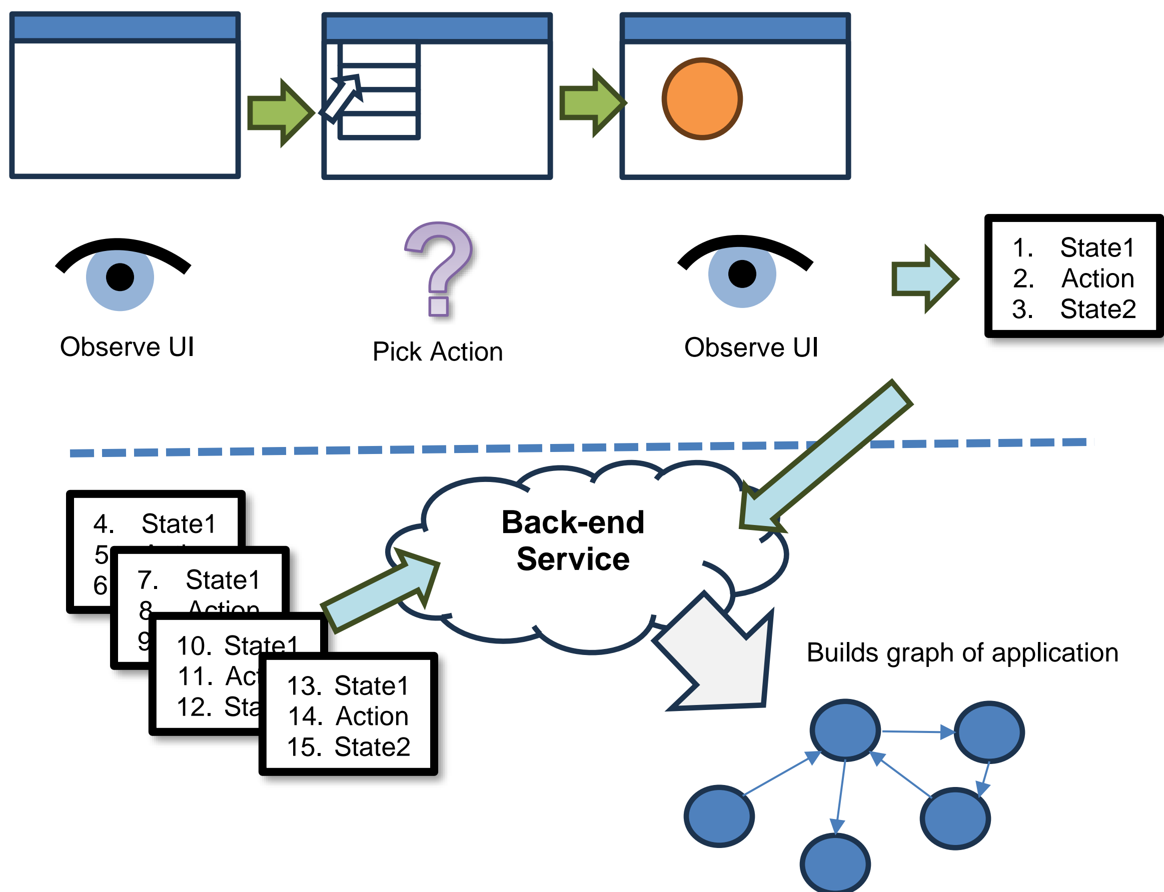


Figure 2 Illustration of agents sending state action sequences to service that builds a graph of application behavior.

On the backend, the service receiving these UI state updates from multiple agents builds a graph that ties state changes to actions. This graph serves as a map to the application behaviors used later after training.

3.2 Collection of rewards from signal, train the predictive model.

A scheduled job collected training rewards from product signal data. A daily process would query the Office telemetry logs and collect events emitted from the Office client while the agents were building the

Copies may not be made or distributed for commercial use

model. Every event carried a machine identifier, the time of the event, and the name. Each event is considered a “reward”. These rewards were compiled in a batch and sent to a service.

For example, a given reward instance might look like:

```
Time: 2022-12-01, MachineId: de734cb4-3c2c-44bb-b077-d6bdf3bcc017, RewardId:
Office.Word.Commanding.Format.BeginBold
```

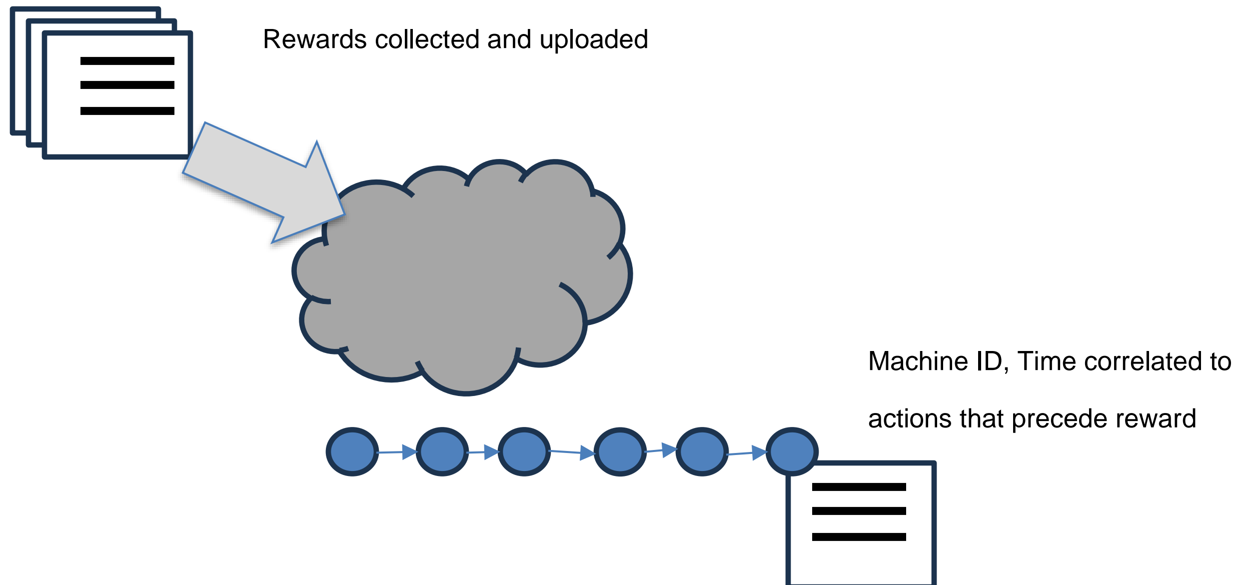


Figure 3 Backend training to correlate rewards with actions that precede them.

On the backend service, a nightly process correlated the rewards with the run that produced them using a combination of the machine id and time. The last several actions in the run prior to the event are collected as a sequence correlated with the reward. After all the sequences are collected for all the rewards, the backend begins training an ML model to predict which sequences of actions predict ought to generate which reward.

Periodically, a table of reward weights that reflect the desired frequency of each reward is uploaded to the service. This allowed us to instruct the service to attempt to produce certain rewards more often than others.

3.3 Targeted navigation based on predictions.

After training, some of the agents are run in a targeted navigation mode where they goal is to reproduce the rewards collected for training. The agent machines contact the backend service and ask for instructions. The service sends them a package with the model built during exploration, the desired reward, and a sequence of actions predicted to produce that reward.

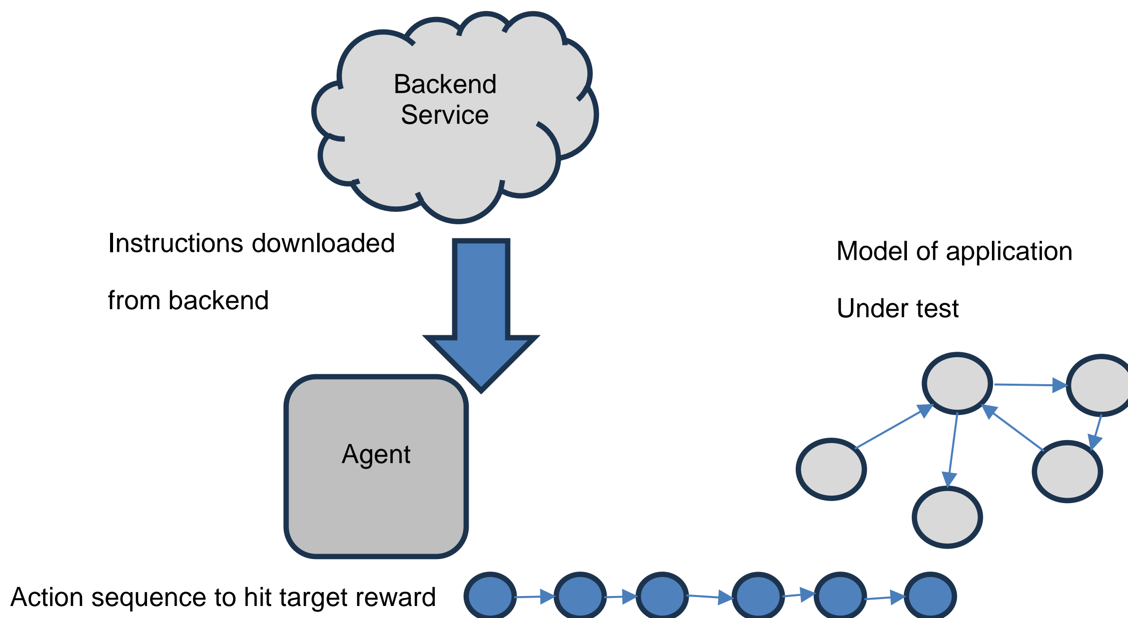


Figure 4 Agent receiving instructions from service for actions to generate reward.

The agent uses the model to build a strategy to get itself to a state where the first action in the sequence would be available. From there, it attempts to complete the sequence. If an action is unavailable, the agent will use the model to predict a sequence from its current state to one where the next action ought to be available.

As it attempts to repeat the sequence, it uploads the same sort of UI state and model sequences gathered during initial exploration, allowing the server to update its model of application behaviors. It also uploads information about whether it can reach the final step in the sequence or decides to abort.

4 Kinds of failure

Our effort started with failures that were easy to identify in the signal. Further on in this document I will elaborate on challenges represented by extracting failures from a telemetry signal, but the main ones we looked at were the following:

4.1 Crashes/hangs/aborts

Microsoft's Windows Error Reporting service collects crash, hang, and abort data from all Windows applications. Microsoft Office collects the crash data and correlates it with the product data by a session id stored with the crash signature. There is a service from the Windows team that allows product teams to automatically receive error reports for new crashes. Further, the Microsoft Office team has a service using anomaly detection to report new crashes that are showing above certain baseline rates, or whose rate of occurrence has changed. There is an existing process for investigating, triaging crashes, with an active team of engineers from each of the Office product teams involved.

This was the easiest class of failure for us to track. Crashes, hangs, and aborts are always considered a bug, simplifying classification. The existing triage and investigation process gave us a group of people to work with that were experts in that failure domain. We focused on crashes because of simplicity. There were challenges still, covered later.

4.2 Explicit errors in the telemetry signal

There are events in the telemetry stream which are marked as errors, making them easy to extract from the data. We did some initial investigation into explicit errors, but challenges around interpretation and purpose of the event motivated us to pull back from that plan.

4.3 Performance Regressions

The performance team in Office has a performance monitoring infrastructure in place that tracks certain key operations and reports if there is any evidence of regressions in how long those operations take. The reporting is based on percentile distributions – such as length of operation at the 95th percentile of sessions.

One of the challenges of this technique is that pre-full production user populations were small enough that it took up to two weeks to collect enough session data to get a meaningful statistical distribution of performance data and detect a regression.

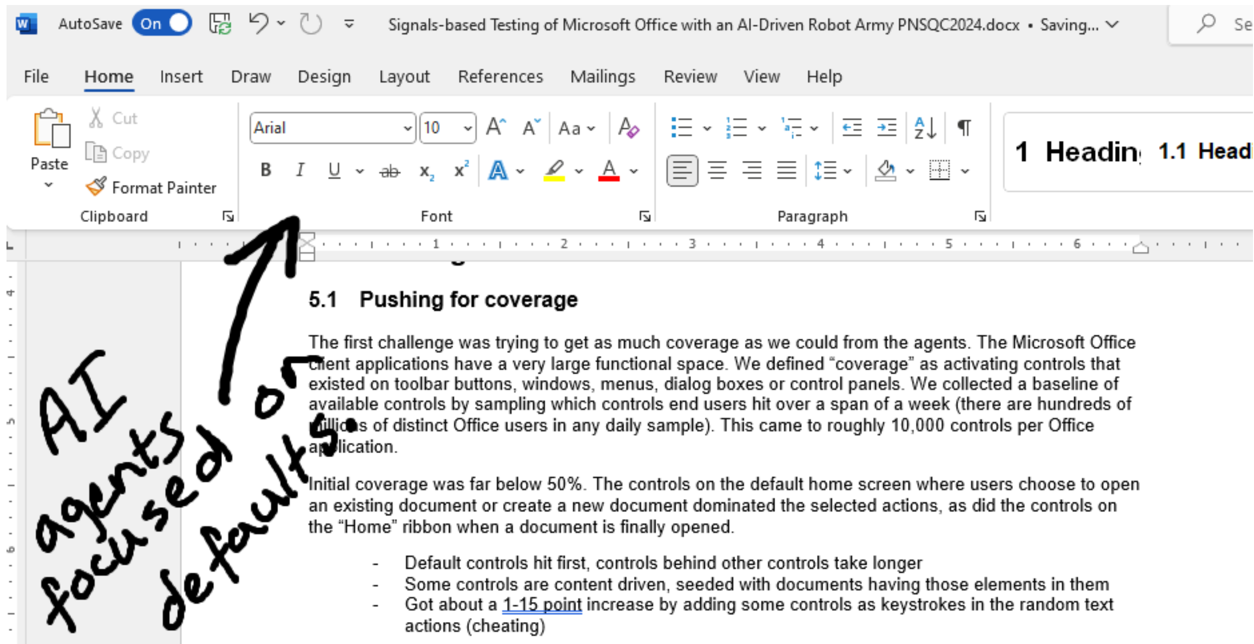
We found that using the synthetic workloads, we could anticipate performance regressions up to a week earlier than our internal rings of users. The robots were “racing the users” to exposing the bugs, and we found they could take the lead. By the time I left in March of 2023, we had demonstrated the technique was effective and were planning to operationalize the reporting.

5 Challenges

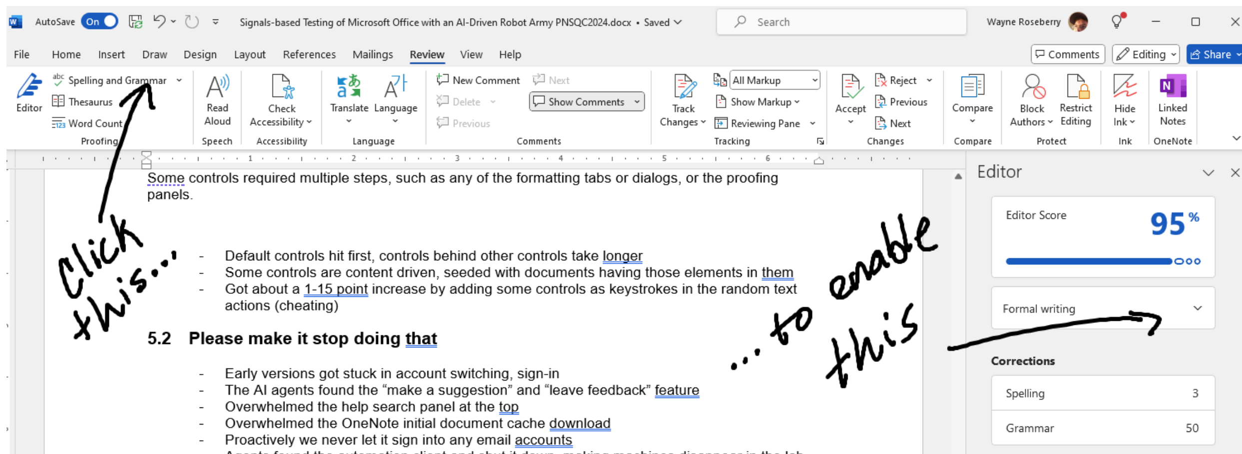
5.1 Pushing for coverage

The first challenge was trying to get as much coverage as we could from the runs. The Microsoft Office client applications have an exceptionally large functional space. We defined “coverage” as activating controls that existed on toolbar buttons, windows, menus, dialog boxes or control panels. We collected a baseline of available controls by sampling which controls end users hit over a span of a week (there are hundreds of millions of distinct Office users in any daily sample). This came to roughly 10,000 controls per Office application.

Initial coverage was far below 50%. The controls on the default home screen where users choose to open an existing document or create a new document dominated the selected actions, as did the controls on the “Home” ribbon when a document is finally opened.



Some controls required multiple steps, such as any of the formatting tabs or dialogs, or the proofing panels. As demonstrated in the screen shots above and below, several of the controls in the spelling and grammar panel are only visible if one first switches to the “Review” tab, and then clicks “Spelling and Grammar” on the ribbon. It took several days for the agents to find controls that lived in menus and dialogs further down the command stack.



There are some controls only visible when certain content exists in the document. Tables, graphic objects, misspelled words, and a variety of other content driven features cause certain menus to appear when the user clicks inside them. The AI agents were not able to create such content easily on their own, so even with weeks of training, they never hit those features.
We addressed that problem with two approaches. The first approach was to seed the canned text strings used by the agents with keystroke accelerators for certain commands. Many of these were document navigation commands (next paragraph, next word, next cell, top of document, bottom of document, select all, etc.) some of these were item insertion (insert graphic, insert shape, insert chart, etc.). In some cases, we identified missing commands from the coverage map and just put in the keystroke equivalent.

The second approach was to seed the test user’s SharePoint site and with documents that had lots of rich content. The agents had already learned to navigate their way through the user’s default document location and would learn to find the test documents. Once the document was open, the agents would

Copies may not be made or distributed for commercial use

learn to find various parts and pieces, and from there the new context-based menus and ribbons would appear, and the agents would add them to the model.

Using this combined approach, we were able to demonstrate that the AI agents hit 70% of the controls which end users were hitting. It took a long-time investigating coverage reports to push the agents to this point, and some of the techniques feel like cheating, but demonstrating that much coverage was what tipped the project over from proof of concept to funding a team to operationalize the approach.

5.2 Please make it stop doing that.

There are features in Office which we try to avoid using during automated runs against live systems. The AI agents tended to find these and fixate on them. In each of these cases, the agent would start hitting the feature, and soon after some member of the product team would notice the traffic, it would disrupt something they were trying to track, or in some cases cause serious problems on a backend service, and they would have to come find us to ask what was going on.

In all these cases it was a matter of configuring the agents to not touch certain controls or applications, but it took a while for us to realize what was happening.

Account Switching and Sign-In: The button to switch accounts and sign-in as a different account is visible on the top right of all Office applications. The problem we hit there is that once the sign-in dialog came up, the bots tended to get stuck there, typing in the wrong password over and over. It was ruining our runs, so we shut that off.

Make a Suggestion and Leave Feedback: This option was available in the upper right corner of Office and is a way for end users to report problems. The team that monitors the internal user feedback for emerging issues saw a flood of strange text (e.g., one of the test strings was “Lisa Simpson is better than Bart Simpson.”) in their reports.

Overwhelming the help search panel: Office has a “get help” box at the top that searches the Office help documentation, as well as offering a shortcut to features in Office. The agents were typing test strings into that box at a rate that was overloading the services deployed for internal Office users.

Overwhelm the OneNote document cache download: The first time a user runs OneNote on their machine, the client checks if they already have a notebook on the server and downloads it to the local machine. The test runs were using a pool of user accounts, but the installation of Office was new on every run, so every run started downloading the prior OneNote notebook for that user, which was getting larger and larger over time. The server deployed for internal use started having outages triggered by notebook downloads far outside the anticipated workload.

Agents kept shutting down the automation client: The test machines have a client that the automation system uses to manage the runs, in particular monitoring when a test is complete and initiating the return of the machines to the system. The agents during training would sometimes discover the automation client and shut it down, making the machine “disappear” from the automation system, which meant an engineer would have to connect to the machine to restore it.

5.3 Random is slow when functional space is massive.

As stated earlier, we eventually managed 70% coverage of the 10,000+ controls per Office client application. The problem was that hitting all those controls took a long time. If one were to start fresh from a given build, reaching the 70% mark took two to three weeks. Some controls were buried so deep in the model, or required such complex conditions to get to them that the system took a very long time to do it.

Another problem is that using weighting and distributions to push the system toward coverage had to spread itself so thin it really did not pay off. If we tried to mimic “real world” percentages we had a very small number of controls that dominated most of the distribution, while the rest would be well under 0.1%.

The only way we were able to hit a lot of the controls in the lesser frequency range was to flatten the probabilities. Defy the real-world percentages and treat everything equally.

With a massive feature set and a user base approaching 1 billion, very small percentages are actually very large numbers. Psychologically we want to believe that percentages such as 0.1% or lower are not worth thinking about, but even numbers that low describe actions occurring hundreds of thousands, even millions of times a day.

We did not find a solution to this problem by the time I left. We knew we wanted to hit target coverage in a shorter period – more like 1-2 days – but we knew that was going to mean substantial changes to the efficiency and directedness of the coverage.

5.4 Bugs are going to be intermittent by nature.

This kind of testing creates a randomness and scale (thousands of machines running 24x7) that is exceptionally good at hitting problems which are occurring in the real world, but which are difficult to reproduce. Crashes, hangs, aborts, and out of memory failures tend to be this way. They tend to be intermittent by nature, as the things that drive them are based on run-time state and conditions not guaranteed to be the same every time the end user does the same action.

This is good, because you want to hit these kinds of failures on your machines where you have access to resources unavailable when the user hits the problem. But such failures are also the most difficult to investigate and fix.

5.5 Definition of failure is not always clear.

Part of the problem we dealt with is that the history of logging errors in Office had not always anticipated us using them as a signal that something went wrong. Many events tagged as errors (which made it easy to find them via a query against the telemetry system) were used by developers for several other purposes. Sometimes it was a way of marking a piece of information the developer wanted to track. Sometimes it was something somebody thought might be an error, but sometimes was not, and they were hoping by marking it that way they could understand the behavior later. Sometimes the error was something that technically went wrong, but for which the product logic had a means to manage and mitigate. Sometimes the error was a normal case of the user doing something wrong and then fixing it on their own. Sometimes the error really meant something bad that we would want to report.

The above is the result of over thirty years of product code. The context and purpose for coding patterns change over time and not everything adapts in a coordinated way. In our case, we were introducing a new purpose where we wanted any error in the telemetry signal to clearly indicate the application believed something was very wrong. This is a problem we had not addressed by the time I was done, but there were other groups in Office working on design pattern proposals with the same goal in mind.

5.6 Understanding what the agent did is difficult.

The way the agents understand the client UI via the `System.Windows.Automation` API is a hierarchical tree of control elements. The agent saved every node it interacted with as a Json block with the process name, control name, automationid, class name, and object type of the object as described by the `System.Windows.Automation` API. The Office developers were not accustomed to thinking about their application in that way, most of them did not know the IDs of the UI components that corresponded to their features (a lot of that constructed in code outside their control). Further, many the objects were missing automation ids, and had names ambiguous with others in the application.

We attempted to address this problem by adding AutomationId fields to controls inside Office to help with disambiguation (also helped with coverage and replay – the agents had trouble using controls with ambiguous object names). We also built tools to visualize the sequences of actions as graphs, particularly

highlighting those that hit problems. This helped give more context to what was going on, helping engineers narrow their investigation.

5.7 Replay is by nature uncertain.

The automation was entirely UI based, where the application and environment state are at its most complex. This meant that not all the states at run-time were guaranteed to be the same on the next run. Some states are different in different runs; different user accounts, MRU list changes, files get changed and saved over time, a variety of other states that live with the user account changes over time and affects the run.

To add to this, the agent is designed to “find” its target state from where it is in the model. This means it may not always be starting from the same state. It also uses this capability to try to correct its path if it ever finds itself on a step where something it expected to be there is not. In this case it evaluates the model and builds a plan to get to the target state to get “unlost.”

These combinations of factors keep replay from being exact. Exact replays are extremely brittle solutions that fail rapidly on complex behaviors, and this whole approach intends to be complex. This means sacrificing exact replay for sake of keeping the system moving.

We had investigations underway to try to achieve as high replay fidelity as possible, but it was always going to be a non-perfect, less than 100% fidelity behavior. The variance is a desired feature of the system. That said, we were finding and fixing fidelity bugs.

Part of the solution to this problem was to focus not so much on exact replay as using data analysis and visualization to show which paths through the application tended toward which failures. We wrote a tool (US patent # 20240111577) that given a specific crash identifier could show how many times the system had hit it and a graph of all the different paths which led to it, ordered by highest hit rate. The idea was not to rely on getting an exact hit on replay but instead to understand the behavior via analysis of the patterns.

5.8 Prioritizing the failures is difficult.

When the agent hits a new crash or other error, it was difficult to know if that problem was worth fixing. It was difficult to anticipate how many people were going to hit that crash when the product was released. The automated runs were hitting thousands of crashes. We would have overwhelmed the product teams if we pushed the product teams to fix all the crashes we found.

Making it even more difficult was that most of the crashes were only seen in the automated runs. For example, there was one feature in Excel that has been there since version 1.0 (not going to tell you which one, sorry, but if you look in books on how to use Excel, it is in there) which is easily available and which the AI agents used a lot, and which they were very good at crashing. Despite this, none of these crashes in this control ever showed up for actual users. Why? I don’t know. There was also a mode that the AI agents were able to get into because the accessibility APIs allowed using controls behind modal dialogs, something the Windows UI blocks. There were crashes occurring in this case because the controls are never used below a modal dialog this way, and consequently we never saw these crashes from end users, only the agents.

During this time, it was about two to three months between the test run and when the product would release to the full market. The internal and pre-release user groups were likewise poor predictors of crashes, hangs, aborts, and other failures seen in the full market. The size of the full user base was so large that their usage patterns overwhelmed everything else. The number of unique crashes captured in the full market was three to four orders of magnitude larger.

But there were crashes that were 1) first seen by the AI agents, 2) eventually seen when the product hit the full market 60 or so days later. The trick was how to predict which ones. We wanted to focus on those likely to show up and also likely to happen often enough to motivate the product team to fix them.

We started an investigation into using binary classification to predict whether the product teams would resolve a given crash FIXED or WON'T FIX, based on call stack patterns in the crash. Call stacks give clues to where in the code the crash occurred, which can also be a rough proxy for a given feature. The Windows Error Reporting service gave us call stacks for end user reported crashes, so we were able to build a model that classified prior call stack entries as predicting either FIXED or WON'T FIX on the bug, and based on make the same prediction for crashes newly discovered by the AI agents. The idea was that if the call stacks were hitting similar parts of the code as prior crashes the product team fixed, then those parts of the call stack must correspond to feature areas more popular or deemed more important to the product team. We were able to demonstrate this technique reduced the number of crashes to examine for fixing from thousands to dozens.

The other technique we used was relying on crashes already in market, but which the product team had been unable to discover the fix. In some cases, product team engineers came to us when they noticed we were hitting bugs they had been chasing for a while and we would help them out trying to understand what was happening.

5.9 People incorrectly believe this approach replaces other testing activities.

People often want to pretend AI and machine learning based tools are far more capable than they really are. When they hear that a tool is on its own learning how to navigate the application space and provide some kind of testing value, they start pushing to use it to replace existing investments in automation and human testing activities. People came to me multiple times pushing me to engage in this sort of replacement. I checked with the manager of the team inventing the tool, and he said the same thing happened to him frequently. We both had the same reasons why the replacement proposal does not hold water.

Automated scenarios and checks: Our existing large suite of automated scenarios and checks were designed to examine specific and exact conditions. They were built to manipulate the product into an exact state and then check specific things to see if anything went wrong.

Professional testers executing scenarios: In addition to developers testing their own code, Office had vendors under contract to cover sets of test scenarios. More open-ended than the automated checks, such testing is still done with a human awareness of what might seem wrong, unexpected, problematic. People observe and assess the application state to notice anything wrong.

The AI agents work in almost exactly the opposite fashion. They have no model of what is correct, what is wrong, what is expected. If the application does something it should not, rather than stop and report a problem, the AI agents would just "learn" that as the expected behavior. The automated agent power in our case was in the unpredictability at large scale. We were relying not on precise observation of what was happening in the application, but instead on our own analysis of broad trends and patterns in the signal data. The tool, while being good at what it did do, did not in any way match either the automated checks for their capability to measure exact expectations and follow strict sequences, nor match the human ability to notice a broad set of possible oddities and unexpected problems while exploring the application.

Conclusion

Throwing an AI-driven robot army at a product as large and complex as Microsoft Office was one of the most fascinating, interesting, challenging, and entertaining acts of testing I have ever tried. It was a crazy experiment in a lot of ways. We did demonstrate that the idea was sound but had a very long way to go as well. We had to wrestle with the massive size of the application, the unpredictable nature of the tool, inefficiency of coverage, and difficulty trying to decide what to do with the failures we found. On the way there, we had to invent new tools, techniques, and processes. Taking it further was going to require much more. It was a unique experience, and one that might have been difficult to do anywhere else. I believe that some of the challenges we encountered will echo for others as advanced innovations in testing tools and technology move rapidly forward.

Copies may not be made or distributed for commercial use

References

Patent; Creators: SHAH; Mitansh Rakesh (Seattle, WA), RAHMANI HANZAKI; Mahdi (Toronto, CA), ROSEBERRY; Wayne Matthias (Redmond, WA), SCHICK; Guilherme Augusto Kusano (New Westminster, CA). Title: SYSTEM AND METHOD FOR DETERMINING CRITICAL SEQUENCES OF ACTIONS CAUSING PREDETERMINED EVENTS DURING APPLICATION OPERATIONS. Patent #: 20240111577. Filed: 2022.

Wayne Roseberry, PNSQC 2017, “Is Your Automation Any Good?”, http://uploads.pnsqc.org/2017/papers/Is-Your-Automation-Any-Good-Wayne-Roseberry.pdf

Jason Arbon, PNSQC 2017, “AI For Software Testing”, http://uploads.pnsqc.org/2017/papers/AI-and-Machine-Learning-for-Testers-Jason-Arbon.pdf

Wayne Roseberry, 2023, “Signals Based Software Testing”, https://waynemroseberry.github.io/2023/05/14/Signals-based-software-testing.html

Windows Error Reporting, https://en.wikipedia.org/wiki/Windows_Error_Reporting

Copies may not be made or distributed for commercial use